Data Structure Assignment #1

소프트웨어학부 20234973 정선빈

Assignment #1-1: Implementation

```c
// Dynamic Array
typedef struct {
    int *array;
    int size;
    int capacity;
} DynamicArray;
```

(1) DynamicArray

- 크기가 동적으로 조정되는 배열
- DynamicArray 구조체로 구현
-

■ Initialization with a specific size
■ Inserting an element (front, back, specific index)
■ Deleting an element (front, back, specific index)
■ Accessing an element at a specific index

1. **초기화** : createDynamicArray 함수를 사용하여 특정 크기로 초기화

```c
DynamicArray* createDynamicArray(int capacity) {
    DynamicArray* dynArr = (DynamicArray*)malloc( size: sizeof(DynamicArray));
    dynArr->array = (int*)malloc( size: capacity * sizeof(int));
    dynArr->size = 0;
    dynArr->capacity = capacity;
    return dynArr;
}
```

2. **삽입** : insertFrontDynamicArray, insertBackDynamicArray, insertAtIndexDynamicArray 의 함수를 사용하여 요소를 배열의 앞, 뒤 또는 특정 인덱스에 삽입

```c
void insertAtIndexDynamicArray(DynamicArray* dynArr, int index, int element) {
    if (index < 0 || index > dynArr->size) {
        // printf("Index out of bounds\n");
        return;
    }
    if (dynArr->size == dynArr->capacity) {
        resizeDynamicArray(dynArr, newCapacity: 2 * dynArr->capacity);
    }
    for (int i = dynArr->size; i > index; i--) {
        dynArr->array[i] = dynArr->array[i - 1];
    }
    dynArr->array[index] = element;
    dynArr->size++;
}
```

```
void insertFrontDynamicArray(DynamicArray* dynArr, int element) {
    if (dynArr->size == dynArr->capacity) {
        resizeDynamicArray(dynArr, newCapacity: 2 * dynArr->capacity);
    }
    for (int i = dynArr->size; i > 0; i--) {
        dynArr->array[i] = dynArr->array[i - 1];
    }
    dynArr->array[0] = element;
    dynArr->size++;
}

void insertBackDynamicArray(DynamicArray* dynArr, int element) {
    if (dynArr->size == dynArr->capacity) {
        resizeDynamicArray(dynArr, newCapacity: 2 * dynArr->capacity);
    }
    dynArr->array[dynArr->size] = element;
    dynArr->size++;
}
```

3. **삭제** : deleteFrontDynamicArray, deleteBackDynamicArray, deleteAtIndexDynamicArray 의 함수를 사용하여 배열의 앞, 뒤 또는 특정 인덱스에서 요소를 삭제

```
void deleteFrontDynamicArray(DynamicArray* dynArr) {
    if (dynArr->size == 0) {
        // printf("Array is empty\n");
        return;
    }
    for (int i = 1; i < dynArr->size; i++) {
        dynArr->array[i - 1] = dynArr->array[i];
    }
    dynArr->size--;
}

void deleteBackDynamicArray(DynamicArray* dynArr) {
    if (dynArr->size == 0) {
        // printf("Array is empty\n");
        return;
    }
    dynArr->size--;
}
```

```
void deleteAtIndexDynamicArray(DynamicArray* dynArr, int index) {
    if (index < 0 || index >= dynArr->size) {
        // printf("Index out of bounds\n");
        return;
    }
    for (int i = index; i < dynArr->size - 1; i++) {
        dynArr->array[i] = dynArr->array[i + 1];
    }
    dynArr->size--;
}
```

4. **접근** : accessAtIndexDynamicArray 함수를 사용하여 배열의 특정 인덱스에 있는 요소에 접근

```
int accessAtIndexDynamicArray(DynamicArray* dynArr, int index) {
    if (index < 0 || index >= dynArr->size) {
        // printf("Index out of bounds\n");
        return -1; // Return a default value or handle error appropriately
    }
    return dynArr->array[index];
}
```

5. **Free**

```
void freeDynamicArray(DynamicArray* dynArr) {
    free(dynArr->array);
    free(dynArr);
}
```

(2) A singly linked list

- 각 노드가 데이터와 다음 노드를 가리키는 포인터를 포함하는 구조
- SinglyLinkedList 구조체로 구현

```
// Singly Linked List
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct {
    Node* head;
    int size;
} SinglyLinkedList;
```

■ Initialization (empty list)
■ Inserting an element (front, back, specific index)
■ Deleting an element (front, back, specific index)
■ Accessing the element at the head

1. **초기화** : createSinglyLinkedList 함수를 사용하여 빈 연결 리스트를 초기화

```
SinglyLinkedList* createSinglyLinkedList() {
    SinglyLinkedList* list = (SinglyLinkedList*)malloc( size: sizeof(SinglyLinkedList));
    list->head = NULL;
    list->size = 0;
    return list;
}
```

2. **삽입** : insertFrontSinglyLinkedList, insertBackSinglyLinkedList, insertAtIndexSinglyLinkedList 의 함수를 사용하여 연결 리스트의 앞, 뒤 또는 특정 인덱스에 요소 삽입

```
void insertFrontSinglyLinkedList(SinglyLinkedList* list, int data) {
    Node* newNode = (Node*)malloc( size: sizeof(Node));
    newNode->data = data;
    newNode->next = list->head;
    list->head = newNode;
    list->size++;
}

void insertBackSinglyLinkedList(SinglyLinkedList* list, int data) {
    Node* newNode = (Node*)malloc( size: sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    if (list->head == NULL) {
        list->head = newNode;
    } else {
        Node* current = list->head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
    list->size++;
}
```

```
void insertAtIndexSinglyLinkedList(SinglyLinkedList* list, int index, int data) {
    if (index < 0 || index > list->size) {
        // printf("Index out of bounds\n");
        return;
    }
    if (index == 0) {
        insertFrontSinglyLinkedList(list, data);
        return;
    }
    Node* newNode = (Node*)malloc( size: sizeof(Node));
    newNode->data = data;
    Node* current = list->head;
    for (int i = 0; i < index - 1; i++) {
        current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
    list->size++;
}
```

3. **삭제** : deleteFrontSinglyLinkedList, deleteBackSinglyLinkedList, deleteAtIndexSinglyLinkedList 등의 함수를 사용하여 연결 리스트의 앞, 뒤 또는 특정 인덱스에서 요소 삭제

```
void deleteFrontSinglyLinkedList(SinglyLinkedList* list) {
    if (list->head == NULL) {
        // printf("List is empty\n");
        return;
    }
    Node* temp = list->head;
    list->head = list->head->next;
    free(temp);
    list->size--;
}
```

```c
void deleteBackSinglyLinkedList(SinglyLinkedList* list) {
    if (list->head == NULL) {
        // printf("List is empty\n");
        return;
    }
    if (list->head->next == NULL) {
        free(list->head);
        list->head = NULL;
        list->size--;
        return;
    }
    Node* current = list->head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    free(current->next);
    current->next = NULL;
    list->size--;
}
```

```c
void deleteAtIndexSinglyLinkedList(SinglyLinkedList* list, int index) {
    if (index < 0 || index >= list->size) {
        // printf("Index out of bounds\n");
        return;
    }
    if (index == 0) {
        deleteFrontSinglyLinkedList(list);
        return;
    }
    Node* current = list->head;
    for (int i = 0; i < index - 1; i++) {
        current = current->next;
    }
    Node* temp = current->next;
    current->next = current->next->next;
    free(temp);
    list->size--;
}
```

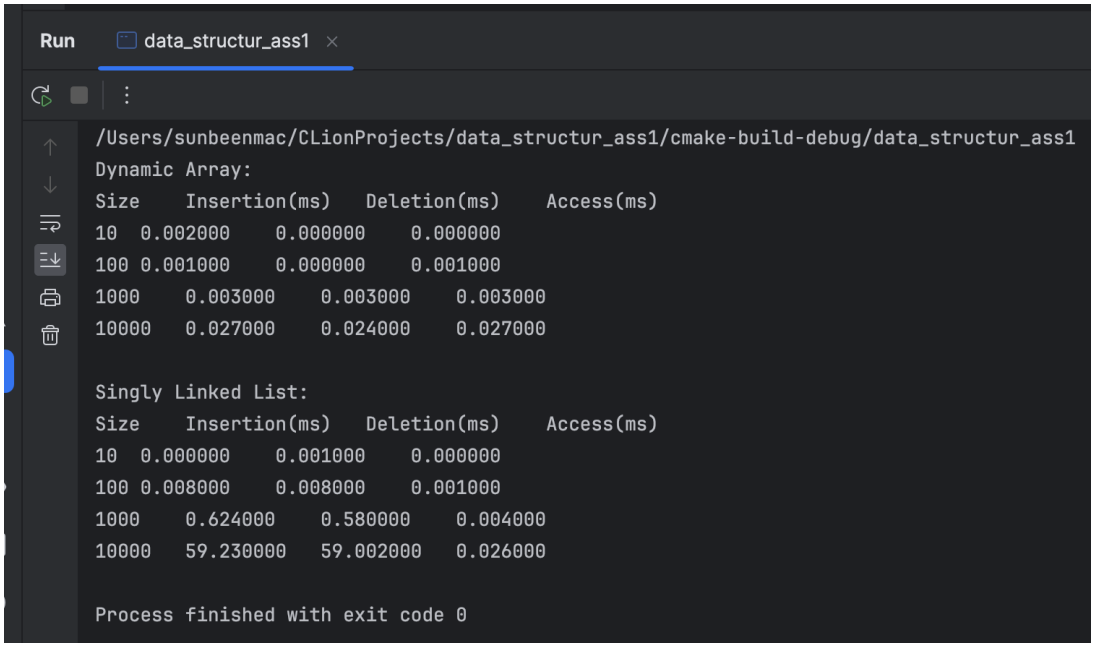
4. **접근** : accessAtIndexSinglyLinkedList 함수를 사용하여 연결 리스트의 헤드에 있는 요소에 접근

```c
int accessAtIndexSinglyLinkedList(SinglyLinkedList* list, int index) {
    if (index < 0 || index >= list->size) {
        // printf("Index out of bounds\n");
        return -1; // Return a default value or handle error appropriately
    }
    Node* current = list->head;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
```

5. **Free**

```c
void freeSinglyLinkedList(SinglyLinkedList* list) {
    Node* current = list->head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
    free(list);
}
```

Assignment #1-2: Time Complexity Analysis



Dynamic Array

1. insertion

- 크기 10에서 1000까지, 삽입 시간은 비교적 일정하게 유지돼있음

-> O(1)의 시간 복잡도 나타냄

- 그러나 크기 10000에서는 삽입 시간이 크게 증가하며, 크기 조정으로 인한 선형 시간 복잡도, 즉 O(n)에 가까운 것으로 추정

2. deletion

- 삽입과 마찬가지로 크기 10에서 1000까지 삭제 시간은 비교적 일정하게 유지됨

-> O(1)의 시간 복잡도 나타냄

- 그러나 크기 10000에서 삭제 시간이 크게 증가하며, 크기 조정으로 인한 선형 시간 복잡도, 즉 O(n)에 가까운 것으로 추정

3. access

- 모든 크기에 대해 접근 시간은 비교적 일정하게 유지되어 있으며, 배열의 요소에 대한 접근에 대해 일관된 O(1)의 시간 복잡도를 가짐.

A singly linked list

1. insertion

- 머리에서 삽입 할 경우 O(1)을 가짐. 크기 10에서 1000까지, 삽입 시간이 점진적으로 증가함

-> 선형 시간 복잡도, O(n)를 가짐.

- 그러나 크기 10000에서는 삽입 시간이 크게 증가하며, 리스트 끝에 도달하기 위한 순회로 인한 이차 시간 복잡도, 즉 O(n^2)에 가까운 것으로 추정됨.

2. deletion

- 삽입과 마찬가지로 크기 10에서 1000까지 삭제 시간은 점진적으로 증가하는 것으로 나타납니다. 선형 시간 복잡도, O(n)를 가짐.

- 그러나 크기 10000에서는 삭제 시간이 크게 증가하며, 리스트 끝에 도달하기 위한 순회로 인한 이차 시간 복잡도, O(n^2)에 가까운 것으로 추정됨.

3. access

- 모든 크기에 대해 접근 시간은 비교적 일정하게 유지되어 있으며, 연결 리스트의 요소에 대한 접근에 대해 일관된 O(1)의 시간 복잡도를 가짐.

결론

(1) dynamic array

- 작은 크기의 경우, 삽입 및 삭제 함수에 대해 O(1)의 시간 복잡도를 가지며, 접근 작업에 대해서도 높은 성능을 보임. 그러나 크기가 커짐에 따라 크기 조정으로 인한 시간 복잡도가 증가하여 삽입 및 삭제 작업에 대한 성능이 저하됨.

(2) a singly linked list

- 작은 크기의 경우, 삽입 및 삭제 함수에 대해 O(n)의 선형 시간 복잡도를 가지며, 크기가 커짐에 따라 순회에 따른 이차 시간 복잡도가 발생하여 성능이 더욱 저하됨. 예외적으로 access은 항상 O(1)의 성능을 보임.

Assignment #1-3: Space Complexity Analysis

(1) Dynamic array

- 동적 배열의 공간 복잡도는 주로 저장된 요소의 수와 배열의 초기 용량에 따라 달라짐.
- 요소 자체를 저장하는 데 필요한 공간은 요소 수(n)에 비례
- 추가적인 메타데이터(배열 용량 등)로 인한 오버헤드가 있음.
- 요소 수가 초기 용량을 넘어가면 동적 배열은 크기를 두 배로 조정합니다. 이 때 추가적인 메모리 할당과 기존 요소의 복사가 발생할 수 있음.
- 따라서 동적 배열의 공간 복잡도는 요소 수(n)에 대해 최악의 경우 O(n)으

로 볼 수 있지만 크기 조정 오버헤드를 고려하면 주기적으로 O(n)의 평균 복잡도로 볼 수 있음.

(1) A singly linked list

- 연결 리스트의 공간 복잡도도 저장된 요소의 수에 따라 결정됨.
- 각 요소(노드)는 데이터를 저장하는 데 필요한 메모리와 다음 노드를 가리키는 포인터를 저장하는 데 필요한 메모리를 사용
- 동적 배열과 달리 연결 리스트는 초기 용량이나 크기 조정의 오버헤드가 없음
- 그러나 연결 리스트는 노드를 서로 연결하는 포인터로 인한 추가적인 오버헤드가 있음
- 따라서 연결 리스트의 공간 복잡도도 요소 수(n)에 대해 O(n)으로 볼 수 있음.

요소 수와의 관계

(1) Dynamic array

- 요소 수(n)와 함께 선형적으로 공간 복잡도가 증가 : (O(n)).

- 요소 수가 초기 용량을 넘어가면 주기적으로 크기 조정이 발생할 수 있으며, 이 때 메모리 사용량이 급격히 증가할 수 있음.

(2) A singly linked list

- 동적 배열과 같이 요소 수(n)에 비례하여 선형적으로 증가(O(n)).

- 그러나 연결 리스트는 크기 조정의 오버헤드가 없으므로 메모리 사용량이 동적 배열보다 안정적임.

Assignment #1-4: Report

동적 배열을 사용하는 시나리오

- 요소의 수가 사전에 알려져 있거나 상대적으로 작을 때
- 요소에 빈번한 임의 접근이 필요한 경우

trade-offs

- 높은 성능을 제공하지만 크기 조정 오버헤드와 메모리 단편화가 발생할 수 있음.
- 연속된 메모리 액세스로 인해 빠른 액세스 시간을 제공함.

연결 리스트를 사용하는 시나리오

- 요소의 수가 크거나 예측이 어려울 때
- 요소에 빈번한 삽입 또는 삭제 작업이 필요한 경우
- 메모리 단편화가 고려되는 시나리오에 사용

trade-offs

- 빠른 삽입 및 삭제 작업을 제공하지만 순회로 인해 접근 시간이 느림.
- 포인터 오버헤드로 인해 요소 당 더 많은 메모리를 사용함.

결론

- 동적 배열과 연결 리스트 중 선택은 데이터의 크기와 예측 가능성, 작업의 빈도, 메모리 제한 및 액세스 패턴과 같은 다양한 요인에 의해 결정
- 동적 배열은 액세스 작업에 대한 성능이 우수하지만 연결 리스트는 삽입 및 삭제 작업에 우수 -> 이러한 작업의 우선순위에 따라 선택이 달라짐.